DRAFT

APPLICATION PROGRAMMER INTERFACE (API) FOR THE DII COE COMMON MESSAGE PROCESSOR, VERSION 1.0.2.4 FOR SUN SOLARIS 2.5.1

April 24, 1997

Prepared for:
Program Manager
Common Hardware Software
Ft. Monmouth, NJ 07703

DRAFT

APPLICATION PROGRAMMER'S INTERFACE MANUAL
FOR THE COMMON OPERATING ENVIRONMENT
COMMON MESSAGE PROCESSOR

1 SCOPE

This document is the Application Programmer=s Interface (API) Manual for the Army recommended Common Operating Environment (COE) Common Message Processor (CMP). This document is designed to provide to the client software programmer information regarding the interfaces between the CMP components and the client application system. Refer to the user's manuals listed in paragraph 3 of this document for detailed instructions for use of the CMP components. A listing of acronyms and their definitions used in this document is contained in Appendix A.

1.1 SYSTEM OVERVIEW

The CMP is the COE's message handling portion of the common support software suite. The CMP consists of the objects and public operations for those objects that support the requirements to handle incoming and outgoing message traffic. Figure 1 shows a single system with integrated CMP components.

The CMP consists of two stand-alone software systems that form the core message handling functions of processing incoming messages, and editing or creating outgoing messages. Other software modules provide the additional functionality of message journaling, message repository, automatic generation, and message data normalization.

The CMP core separate software modules, the Inbound Message Processing (JMAPS) and the Outbound Message Generation (JMPS) are augmented by Army produced software which provide the additional functionality required in the Software Requirements Specification (SRS).

2. GCCS Message Module Application Programmer Interfaces (APIs)

This next section will contain a description of the required Message block APIs. These APIs are derived based on the Global Command and Control Systems (GCCS) Common Operating Environment (COE) Message Processing SRS.

2.1 **Inbound Message Processing**

2.1.1 **Get_Msg<MSG,Status>**

2.1.1.1 Purpose

The concept of software functional independence requires that each functional area be defined and operate as an independent module. This concept is also applied within the individual modules to provide tailoring of functionality by a using system. The Message Processing module is constructed so as to provide isolation on either side by either a file or queue. On the inbound side, the message processing module expects the communications module to pre-process inbound messages removing protocol wrapper and passing only message header and body content to the CMP. The CMP accesses these messages using the Get_Msg API which consist of:

2.1.1.2 Get_Msg Inputs

There are no defined Input parameters.

2.1.1.3 Get_Msg Outputs

The outputs of this API are the received message <MSG> from the GCCS COE Communications Block and the status <status> of the Message.

2.1.1.4 CMP Defined API for Get_Msg Functionality

The CMP defines the <disc> process. This process will wait to receive a message from any Communications source into its user definable inbound directory. Current implementations are to receive messages placed in a defined directory; to receive VMF messages from TCP/IP Port 10000, and to receive messages from DII Communications Server. Finally, The CMP utilizes the defined public API from the DII Communications package known as Get_USMTF_DATA(Id,Header,Msg). The input parameters are the unique id. The output parameters are the Header data and the Message. For complete information, please refer to the DII Communications API document.

2.1.2 Determine_Route(Msg_Header,Msg,Route_Ind)

2.1.2.1 Purpose

Once the CMP inbound processing module receives a message from the communications block there must be some decisions made as to where the information is to be sent. Based upon the decision this API can route the message to one, or more, of five defined areas: Interactive Processing, SRI processing, Bit Oriented Message to Character Oriented Message Translation, a Functional user, or to Inbound Parsing.

2.1.2.2 Determine_Route Inputs

The Inputs to this API are the Message and associated header information.

2.1.2.3 Determine_Route Outputs

The Output of this API is the Route_Ind, which specifies where the message should be routed.

2.1.2.4 CMP Defined API for Determine_Route Functionality

The CMP defines the <disc> process. This process will wait to receive a message from any Communications source into its user definable inbound directory. Then it will determine if the message should be routed to BOM to COM processing, Inbound Message Processing, etc. Additionally, the can filter on what VMF messages to be processed by adding them to be included/excluded in the VMF_MSGPOOL file. The VMF_MSGPOOL file is a sequential list of VMF message ids in the form of <FAD><MSGNUM), where FAD is the Functional Area Designator and MSGNUM is the Message number.

2.1.3 **Put_to_OMJ (Hdr,Msg,Status)**

2.1.3.1 Purpose

Inbound messages and those released to the communications block for transmission are stored in the operational message journal. Data in this journal is user retrievable for viewing and/or modification and re-release as a new message. CMP uses the Put_to_OMJ API for insertion of both inbound and outbound messages into this journal.

2.1.3.2 Put_To_OMJ Inputs

The inputs to this API are the message header and the message.

2.1.3.3 Put_To_OMJ Outputs

The outputs of this API is a status indicating success or failure.

2.1.3.4 CMP Defined API for Put_To_OMJ Functionality

```
long rpc_put_msg
(
  [in] HEADER_INFO *header_info,
  [in] char IO,
  [in] TEXT_FILE *Msg
);
```

<u>Parameters</u>

Input -

header_info	- message header information.
IO	- Inbound/outbound message.
Msg	- Message strings.

Output -

None.

Return Values -

This procedure returns the token number(1-n) upon successfully journaling the message, otherwise it returns zero(0) for failure.

2.1.4 **Message Profiling (Criteria,List_Of_Msgs)**

2.1.4.1 Purpose

The users of the CMP have defined a requirement to be able review a list of messages contained in the OMJ and determine messages of interest to them based on a message profile. CMP has interpreted and implemented this requirement through extraction of user definable message parameters, which are sortable and refinable, and presented as a directory output. Selection of a directory entry provides the user with the capability to review and/or retrieve the message for editing and submission as a new message or additional processing. The Message_Profile API consist of:

2.1.4.2 Message_Profile Inputs

The API inputs are the user specified filtering criteria.

2.1.4.3 Message_Profile Outputs

The API output is the list of messages which meet the specified criteria.

2.1.4.4 CMP Defined API for Message_Profile Functionality

Currently, there are two methods which can be used to perform Message profiling, the first is the AMHS, reference the Jet Propulsion Lab AMHS API document for a complete description. The second is associated with the CMP Journaling component as described below:

```
long rpc_filter_journal
(
  [in,string] char *criList,
  [out] FILTER_LIST **Filter_List
);
```

Parameters

Input

criList	- Filtering criteria

Output

Filter_List	- A list of header information returned based on the input criteria.

Return Values

1	- The returned Filter_List is valid
0	- The call failed to produce a valid Filter_List.

2.1.5 **Message_Validation (Msg,Results)**

2.1.5.1 Purpose

Message validation is the process of comparing the message structure and field contents with the message definition contained in the appropriate message definition documentation. Validation, as defined in the CMP, compares the inbound or newly generated message to insure that all required sets and fields are present and that fields contain data structured in compliance with allowable syntax rules. The CMP provides a user the with the option to determine what, if any, level of validation is to occur against a message being processed. Messages validation parameters may be specified based on message type or originator. The message identification and level of validation for inbound messages are contained as part of the query statement defining specific processes to be taken against the message. The user may specify no validation, full validation, or partial (checking of fields which are to be parsed) validation. The Message_Validation components are:

2.1.5.2 Message_Validation Inputs

The input to this API is the message.

2.1.5.3 Message_Validation Outputs

The output to the API is either a validated message or a list of structural and field level errors.

2.1.5.4 CMP Defined API for Message_Validation Functionality

CMP provides either submitting message for validation through the mtfval API or into JMAPS for validation.

mtfval accepts the following optional command-line arguments:

-v	If mtfval is reading from files named on the command line, the name of each input file is printed before the error report for that file. File names are printed for both correct and incorrect messages.
-o	(outputfile) All output is written to outputfile instead of the standard output.

C++ Programmers API

The MTFMsg class contains a complete set of C++ programmer API**s** for a user to validate a message including Field level,set level and structural notation validation APIs.

In the MTFMsg class the following member exist

virtual Bitset parse (const ValidateLvl&)

This routine validates a given message.

Input
   ValidateLvl - Specifies the validation level (field,set,snval, or ALL)

Output
   None.

Returned Values
   A Bitset containing the detected errors.

2.1.6 **Parse_Msg(Msg,Specified_Data,Extracted_Items,Status)**

2.1.6.1 Purpose

Message parsing is the process of decomposing an inbound data stream into component parts. The CMP implements this concept and adds the capability for a user to specify which component parts are of interest. The CMP then extracts only the parts of interest and provides them as an ASCII string to a user specified location/process. User definition of which components are to be extracted and the output location are specified using a CMP query language. The Parse_Msg API components are:

2.1.6.2 Parse_Msg Inputs

The parse_msg inputs are the message<MSG>, and the specified data<Specified Data> to be extracted.

2.1.6.3 Parse_Msg Outputs

The parse_msg output will be the extracted data<Extracted Items> and a status indicating and problems with the message<Status>.

2.1.6.4 CMP Defined API for Parse_Msg Functionality

The CMP provides two API**s** that provide equivalent functionality to the Parse_Msg API. The first is submitting a message to The Inbound Message Processing component (JMAPS) against a defined query will extract data. The second is the mtfxtract API. Its defined as follows:

mtfxtract -qf <query_file><Msg_file>, where query file contains a specified JQL and Msg_file contains a message.

C++ Programmers API
   Within the AQLShell class, the following member exist:

```
ShellReport *execute (MTFMsg&, const ValidateLvl = (*ValidateLvl*)nil));
```

Input
MTFMsg& - Contains the message.
ValidateLvl - What level to validate the message on (field,set,structural, ALL)

OutPut
None

Returned Value:
A ShellReport showing the parsed message.

2.1.7 **Application_Interface (Functional_User,Selected_Items, Data Results)**

2.1.7.1 Purpose

Once an inbound message has been validated, if required, and parsed the extracted data will be provided to a user specified location/process. This location/process may be to the normalization module for additional processing, to a database interface for posting to a database, to a user for interactive processing/viewing, or to a process for additional processing (like posting to a map). This API provides that interface where the data will be handed to the requester and is comprised of:

2.1.7.2 App_Int Inputs

The inputs to this API are the Functional_User (An application to receive the data) and the items the items the user wishes to receive (Selected Items).

2.1.7.3 App_Int Outputs

The API output is the <Data Results> sent to the functional user.

2.1.7.4 CMP Defined API for App_Int Functionality

The CMP provides the App_Int functionality through the use of the Inbound Message Processing (JMAPS Query Language (JQL)) route command:

```
route msgtype query q-name [host hostname] cmd cmd-string;
```

The msgtype is the name of the message type being routed. To have any meaningful effect, it should be one of the message types defined in the current messaging standard that the user is utilizing.

The query section is mandatory. It specifies the name of the query or shell that will be run on each incoming message of the specified message type.

The host section is optional. The hostname specifies the name of the machine that will run the client application receiving the query results. If this section is omitted, JMAPS assumes the host is local host.

The cmd section is mandatory. The cmd-string specifies the name of the client application that will be executed each time a msgtype message arrives. It consists of any sequence of characters, not including a semicolon (";"). In this string, the case of alphabetic characters is significant; for example, /usr/local/isum/isum will not have the same effect as /usr/local/isum/ISUM.

The command string will eventually be executed by the Bourne shell shell, and so it is unwise to write a command string that includes characters having special significance to the shell.

There may be several routing definitions for the same message type. JMAPS will execute the behavior specified for each routing definition whenever it receives a message of that type. The order of execution of the routing behaviors is undefined.

2.2 **Outbound Message Generation**

2.2.1 **Message Generation - Interactive Create_Msg_Int (Msg_Type,Msg)**

2.2.1.1 Purpose

Interactive message generation supports both direct data input from a keyboard and computer assisted entries for selected data fields. The user may specify autofill, with defaulted data, for selected fields while retaining the ability to over-ride the entry from the keyboard. The user can also access a Data Item (DI) code from a menu for posting to the data field. Once complete, the message may be submitted directly to the communications block for transmission, saved to a temporary work folder for retrieval and continued processing or saved to a temporary work folder for coordination and later release. Components of the Create_Msg_Int are:

2.2.1.2 Create_Msg_Int Inputs

The Input to this API is the type of message template that the user wishes to interactively fill.

2.2.1.3 Create_Msg_Int Outputs

The output of this API is the requested message template.

2.2.1.4 CMP Defined API for Create_Msg_Int Functionality

The CMP API to access the equivalent functionality is the following:

```
jmps <filename> where filename specifies a non-existing file.
```

The user can access this API by using the CMP Message Handler and selecting the option File,New from the menu. The initial jmps messaging menu will be based on the current message set selected.

C++ APIs/Classes

Class Manager contains the following member:

void importFile (const Vstring&, ifstream&)

Input Parameters

Vstring& - The name of the file.

Output Parameters

ifstream& - The opened file descriptor.

Returned Values

None.

2.2.2 **Message Generation - Automatic Create_Msg_Auto (Msg_Type,Msg)**

2.2.2.1 Purpose

CMP provides the capability to automatically generate and release a message based on inputs from an external process. Examples of these external processes might be the result of a query to a database or satisfaction of a Standing Request for Information (SRI). Data for population of the desired message is provided to an SQL shell script which contains directions as to the message recipient(s), message ID, and structural information. The completed message is forward to either a releaser, if specified by the user, or to the communications block for transmission. The Create_Msg_Auto consist of:

2.2.2.2 Create_Msg_Auto Inputs

The Input to this API is the type of message template that the user wishes to automatically fill.

2.2.2.3 Create_Msg_Auto Outputs

The output of this API is the requested message template.

2.2.2.4 CMP Defined API for Create_Msg_Auto Functionality

The CMP API to access the equivalent functionality are the following:

mdlmap <mdl_script> <data>,where mdl_script is the defined meta language template, and <data> is the data to be autofilled

The user can access this API by using the CMP Message Handler and selecting the option Autofill,<Msg_Type> from the menu.

C APIs

Several messages have the autofill functionality added to them. An example of the

API is

```
boolean_type c400_autifill(
        [in] handle_t bh,
        [out] c400_autofill_struct **ptr_to_c400);
```

Input Parameter

handle_t bh - A binding handle (In Distributed Computing Environment (DCE)) terms.

Output Parameters

c400_autofill_struct **ptr_to_c400 - A pointer to the autofilled message structure

Returned Values

1 (TRUE) for Success
0 (FALSE) for Failure

2.2.3 **Hdr_Fmt_Selection (List_Of_Destinations,Hdr_Type)**

2.2.3.1 Purpose

To provide the user with a mechanism to complete the message header. Ultimately, the message block will be responsible for providing the Aheader data@ while the Communications block will make the determination as to what type of header to add to a given message based on its destination.

2.2.3.2 Hdr_Fmt_Selection Inputs

The API Input will be the destination information<List_Of_Destination>.

2.2.3.3 Hdr_Fmt_Selection Output

The API output will be the message header that is required for each destination.

2.2.3.4 CMP Defined API for Hdr_Fmt_Selection Functionality

CMP currently offers two methods equivalent to this API. The first is using the header selection from the JMPS GUI. This interface allows a user to define a specific header, i.e. DD173,ULP,JANAP,etc. The second and preferred method is to use the create header option from the CMP User Interface (CUI) GUI. This option creates a Communications free header (creates the header data elements) to be passed to the Communications Block for actual Message Header assembly

C Programmers API utilized in the CUI

```
Widget Create_Header_Selection_Popup (Widget parent,char *hdr_path,
                                      char msg_path);
```

Input Parameters
- Widget parent - The parent window (widget)
- char *hdr_path - The path to the generic header file.
- char msg_path - The path to the message file.

Output Parameters
- None.

Returned Values
- The display (Widget) of the header selections.

2.2.4 **Message Coordination (Msg,Memo,Dest,Ack/Nacks)**

2.2.4.1 Purpose

Users of the CMP require the ability to release messages directly to the communications block for transmission upon completion of message generation or to coordinate the message with a message release authority prior to release. The CMP accommodates the coordination task through placement of the initial message into a Atemporary working folder@ where it may be retrieved, viewed, modified, and ultimately released for transmission. Components of the Message Coordination API are:

2.2.4.2 Message Coordination Input

This API inputs will be the message<Msg>, an optional memo<Memo>, the destination of the person or persons to coordinate with<Dest>, and whether or not acknowledges are required<Acks/Nacks>.

2.2.4.3 Message Coordination Output

The output of this API will be acknowledgment that each destination has coordinated.

2.2.4.4 CMP Defined API for Message Coordination Functionality

This function is currently not defined within the CMP.

2.2.5 **Message Release Authority (Hdr,Msg,Status)**

2.2.5.1 Purpose

The Message Release Authority (MRA) provides the mechanism to control releasing of messages. Decision steps are review of the message, which is held in the temporary working folder, modification of the data content if required/desired, attachment of a memorandum, and/or release of the message to the communications block for transmission. The Message Release Authority API consist of:

2.2.5.2 Message Release Authority Inputs

This API Inputs are the Header information (Hdr) and the message data<Msg>.

2.2.5.3 Message Release Authority Outputs

The output of this API is a status<Status> indication release or rejection.

2.2.5.4 CMP Defined API for Message Release Functionality

CMP currently defines MRA at the individual user level. Each user of the CMP Message Handler has the ability to ASend@ a message. An attempt will be made to Journal this message. If the user is granted privilege to Journal then he will succeed, else fail. In addition, a copy of the message is placed in the System/output directory of the CMH. A user may wish to implement a single point MRA utilizing this information.

C API**s**

void send_msg (Widget active_list, char *msg_hdr_path)

Input Parameters

Widget active_list - The parent display (widget).
Char *msg_hdr_path - A Pointer to the current message of interests path.

Output Parameters

None.

Returned Values

None.

2.3 **Support Services**

2.3.1 **Configure_CMP**

2.3.1.1 Purpose

This API controls configuration parameters all of the CMP components.

Specifically, these are:
JMAPS,JMPS,JOURNALING,NORMALIZATION,CMH,COM/MSG ISC.

2.3.1.2 Configure_CMP Inputs

There are no inputs to this process that the user specifies.

2.3.1.3 Configure_CMP Outputs

The outputs will be a success/fail indication for each component being configured.

2.3.1.4 CMP Defined API for Configure_CMP Functionality

All CMP components will be installed and configured as part of the JMCIS CMP segment install and configuration.

2.3.2 Error Handling (Msg)

2.3.2.1 Purpose

The CMP must provide a mechanism for monitoring messages transactions to other processes and in the case of a malfunction a medium by which these malfunctions can be conveyed to the transaction originator. Specifically, areas which require monitoring and error handling are rejection of a message by the communications block, notice of an incomplete message in the communications block, routing of incorrect messages to a user for interactive handling (this includes plain text messages) and system level status information. The CMP provides this mechanism using the Error Handling API which consist of:

2.3.2.2 Error Handling Inputs

The input to this API will be the message.

2.3.2.3 Error Handling Outputs

The output of this API will be to display the message for interactive disposition.

2.3.2.4 CMP Defined API for Error Handling Functionality

This API is provided by the Inbound Message Processing functionality. A JQL query can be registered to detect messages with user specified levels of validation. The user can choose to validate only the data that a process is extracting, the message, or the message and the header. An invalid message can be routed to the Outbound Message Generation (JMPS) API for interactive disposition.

C++ API**s**/Classes

void Bitset errors() const;

Input Parameters
 None.

Output Parameters
 None.

Returned Values
 Bitset - A bitset containing the errors found in the Message.

2.3.3 **put_to_audit (Msg,Audit_Type,Status)**

2.3.3.1 Purpose

The CMP will rely on a system level audit file to retain security related audit data. The CMP will monitor and validate security functions and hand off specified security related status/request to a system level interface. The CMP API consist of the following:

2.3.3.2 put_to_audit Inputs

The inputs to this API are the message<Msg> and the type of auditable event<Audit_Type>.

2.3.3.3 put_to_audit Outputs

The output of this API is a status indicating success/fail..

2.3.3.4 CMP Defined API for put_to_audit Functionality

The definition of this API is defined outside the scope of the CMP. The CMP provides the ability for its auditable events to be utilized by such an API if defined. The journaling server provides the following API to display Operational Messaging Journal audited events.

```
long rpc_display_audit ([in]TOKEN_t token,
                        [in]REVISION_t Rev,
                        [out]AuditLogType *AuditList)
```

Input Parameters
 TOKEN_t token - The token number of the interested message.
 REVISION_t Rev - The revision number of the interested message.

Output Parameters
 AuditLogType *AuditList - The list of audited events for this message.

Returned Values

1 - Indicates Success.
0 - Indicates Failure.

2.3.4 **Normalize (src,dest,type,in_value,out_value,status)**

2.3.4.1 Purpose

Developers of the CMP realize that integrating systems are prone to store and process information in forms other than that defined by a MTF message standard. The developers have therefore provided a mechanism where data can be translated from a form used by the host system into that defined as a FFIRN/FUD or vice versa. Use of this function is optional and if used it is the responsibility of the using system to provide the normalization table with translation instructions which are representative of data desired by that system. For inbound messages, this translation of data occurs after parsing and prior to the hand off to a process. This API also allows a user to perform both data aliasing and data conversion. The Normalize API consist of:

2.3.4.2 Normalize Input

The API inputs are the data source<src>,data dest<dst>, conversion type<type>, the input value<in_value>, and the normalized output value,<out_value>.

2.3.4.3 Normalize Output

The output of this API is a status indicating success/failure.

2.3.4.4 CMP Defined API for Normalize Functionality

The CMP provides the following API:

```
error_code_type normalize          (cur_bfa,
                                    other_bfa,
                                    rpc_or_msg,
                                    in_or_out,
                                    op,
                                    msg_kind,
                                    input,
                                    output)
```

bfa_type	cur_bfa;	bfa calling this procedure (source for outbound data and destination for inbound data)
bfa_type	other_bfa;	if "in" data, then this is the source BFA, if "out" data, then this parameter=s value is the

		destination BFA
rpc_or_msg_type	rpc_or_msg;	data originated from rpc or msg
in_or_out_type	in_or_out;	data going out from db or data coming in from rpc or msg
op_type	op;	type of normalization to perform
msg_type	msg_kind;	type of message data originated from, S201, S309 etc.
char	*input;	pointer to data to be normalized
char	**output;	pointer to normalized data, will always be in string format

2.3.5 **VMF/COM (vmf_msg,com_msg,status)**

2.3.5.1 Purpose

The Department of Defense (DoD) is migrating more and more to the use Variable Message Format (VMF) for the exchange of information between automated systems. While processing bit data is more efficient, the one draw back to today=s consumer is the desire to allow a human to read and understand the data stream. To accommodate users that wish to exchange information using the VMF methodology, the CMP has implemented a process that translates VMF traffic into a human readable Character Oriented Message (COM) and vice-versa. This also provides for automated processing by existing systems. The VMF/COM API consist of:

2.3.5.2 VMF/COM Inputs

If you are converting from the bit-oriented representation to the Character-Oriented representation then the input is the Variable Message Format (VMF) message.

If you are converting from the Character-Oriented representation to the VMF representation, then the input is the Character-Oriented Message (com_msg)

2.3.5.3 VMF/COM Outputs

If you are converting from the bit-oriented representation to the Character-Oriented representation then the output is the Character-Oriented Message (com_msg)

If you are converting from the Character-Oriented representation to the Bit-Oriented representation, then the input is the Bit-Oriented Message (vmf_msg).

2.3.5.4 CMP Defined API for VMF/COM Functionality

CMP provides the following API**s**:
vmf2mtf(vmf_msg,com_msg,status), where
    vmf_msg - The Bit-Oriented Message
    com_msg - The Character-Oriented Message representation of the bom_msg.
    status -   The status of success/Fail

mtf2vmf (com_msg,bom_msg,status)
    com_msg - The Character-Oriented Message representation of the bom_msg.
    vmf_msg - The Bit-Oriented Message
    status -   The status of success/Fail

If a user chooses to use the disc component and the CMP Message handler; then both these cases are seamlessly handled for the user as part of receiving and sending a message.

2.3.6 **Message_Validation (msg,results)**

2.3.6.1 Purpose

Certification of message processing software is required by the Joint Interoperability and Engineering Organization to prove that systems using the software will exchange information that is readable and usable by recipients of that information. This is termed Acompliance to standards@. In this case, the CMP is designed to operate and comply with multiple standards. The intent of the CMP is to be able to process message which are generated by Joint or Service unique standards. The only stipulation placed by the CMP is that the supported standard be structured in the same manner as the USMTF Central Database System (CDBS) scheme. For messages structured in this manner, the CMP will generate a message file capable of defining both content and structure of the various messages processed. The CMP is therefore capable of determining whether a message processed by it is in conformance with it**s** defining standard. The Message_Validation API may be used by both the inbound and message generation sides of the CMP. The API consist of:

2.3.6.2 Message_Validation Inputs

The input to this API is the message.

2.3.6.3 Message_Validation Outputs

The output to this API is the results, either no errors, or field/structural errors.

2.3.6.4 CMP Defined API for Message_Validation Functionality

The CMP defines the following API:

mtfval -o <error_file> msg_file, where error file allows a user to specify the storage
location of errors found, and the message<msg_file>

Submitting a message to the Inbound Message Generation (JMAPS) with a JQL query defined to specify errors:

select * validate all format errors;

will provide the same results as the mtfval API. In addition the user can control the validation levels. Finally jmps <file_name> will allow a user to view a message with all field level and structural level errors displayed.

2.3.6.5 C++ APIs/Classes

In the MTFMsg class the following member exist

virtual Bitset parse (const ValidateLvl&)

This routine validates a given message.

Input

ValidateLvl - Specifies the validation level (field,set,snval, or ALL)

Output

None.

Returned Values

A Bitset containing the detected errors.

2.3.7 **Multisection Messaging_In (msg_sections,status) Multisection_Messaging_Out(msg)**

2.3.7.1 Purpose

This API allows outgoing message to be section for transmission and allows reassembly as part of the inbound process.

2.3.7.2 Multisection Messaging Inputs

The API input is the message sections for an inbound message. For an outbound message, it will be the message to be sectioned.

2.3.7.3 Multisection Messaging Outputs

The API output will either be a complete message or a partial message with a status indicating completion for the inbound path. For the outbound path, the message will be section and a status returned if there are any problems.

2.3.7.4 CMP Defined API for Multisection Messaging Functionality

Although CMP currently supports handling of incomplete messages, i.e. messages mission sections can be processed. The additional functionality should and will be handled by the Communications block.

C++ API**s**/Classes

In the JANAP128 class, the following API exist to process inbound messages.

bool sectioned (void)

Input Parameters

None.

Output Parameters

None.

Returned Values

1 - To indicate that a section message (SECTION) directive has been encountered.

0 - Indicates this message is not sectioned.

2.3.8 Message Annotation (msg,memo,status)

2.3.8.1 Purpose

Part of message coordination is the ability to comment on the content of the message under review or to attach additional information explaining rationale for the message. Users entering data such as this must be assured that information pertaining to, but not part of the message, will remain within the originating system. To satisfy this user requirement, the CMP will provide a means to annotate, or attach a memorandum, to the base message. The API consist of:

2.3.8.2 Message Annotation Inputs

The inputs to this API are the message (msg) and the associated memo<memo>.

2.3.8.3 Message Annotation Outputs

The output of this API is a status indicating success or failure.

2.3.8.4 CMP Defined API for Message Annotation Functionality

The CMP provides the following API**s** to handle this capability:

```
long rpc_add_memo
(
  [in] long token,
  [in] long revision,
  [out] TEXT_FILE *Memo
);
```

Parameters

Input

token	- Message token ID
revision	- Message revision number

Output

None

Return Values

1	- Add memo ok.
0	- Error in adding memo.

Description

The *rpc_add_memo()* routine adds a memo to a message header. The *memo* field in the header is set to *Y* to indicate there is a memo file attached to the header.

```
long rpc_delete_memo
(
  [in] long token,
  [in] long revision
);
```

Parameters

Input

token	- Message token ID
revision	- Message revision number

Output

None

Return Values

1	- Delete memo ok

0	- Error in deleting memo

<u>Description</u>

The *rpc_delete_memo()* routine deletes a memo from a message header. The *memo* field in the header is set to *N* to indicate that the memo is deleted.

```
long rpc_modify_memo
(
  [in] long token,
  [in] long revision,
  [in]TEXT_FILE *modifyMemo
);
```

<u>Parameters</u>

Input

token	- Message token ID
revision	- Message revision number

Output

None

<u>Return Values</u>

1	- Modify memo ok
0	- Error in modifying memo

<u>Description</u>

The *rpc_modify_memo()* routine modifies a message memo.

```
long rpc_display_memo
(
  [in] long token,
  [in] long revision,
  [out] TEXT_FILE **Memo
);
```

<u>Parameters</u>

Input

token	- Message token ID
revision	- Message revision number

Output

Memo	- Message memo to be returned.

Return Values

1	- Get message memo ok.
0	- Error in getting message memo.

Description

The *rpc_display_memo()* routine gets a message memo and sends it back to the caller.

2.3.9 **Journaled_Message_Retransmission (in_msg,out_msg)**

2.3.9.1 Purpose

Users of the CMP have expressed a desire to be able to store a copy of all incoming and outgoing messages into a file that is accessible and usable by users of the system. Specifically, the users have expressed a desire to be able to retrieve a message from such a file, be able to modify header and/or body of the message and release the modified message as a new message. To accommodate the requirement, the CMP has implemented the Journaled_Message_Retransmission API. The API consist of:

2.3.9.2 Journaled_Message_Retransmission Inputs

The inputs to this API are the selected message to be extracted from the journal<in_msg>.

2.3.9.3 Journaled_Message_Retransmission Outputs

The output to this API is the out_going message<out_msg>.

2.3.9.4 CMP Defined API for Journaled_Message_Retransmission Functionality

The CMP implements this functionality through three API**s** rpc_display_msg, jmps and rpc_put_msg.

First,

```
long rpc_display_msg
(
  [in] long token,
  [in] long revision,
  [out]HEADER_INFO **HeaderInfo,
  [out] TEXT_FILE **Msg
);
```

Parameters

Input

token	- Message token ID
revision	- Message revision number

Output

HeaderInfo	- Header info to be send back to the caller.
Msg	- Message to be sent back to the caller.

Return Values

1	- get message ok.
0	- get message failed.

Description

The *rpc_display_msg()* routine returns the message header and the message text back to the caller.

Secondly
jmps <filename>, where filename is the message returned from the rpc_display_msg API call.

Finally,

long rpc_put_msg
```
(
  [in] HEADER_INFO *header_info,
  [in] char IO,
  [in] TEXT_FILE *Msg
);
```

Parameters
Input -

header_info	- message header information.
IO	- Inbound/outbound message.
Msg	- Message strings.

Output -
None

The Above API**s** are seamlessly implemented when the user selects a message from the Inbound list of the CMP Message handler, then selects the edit function from the CMP Message handler, Finally selects send from the CMP Message handler.

2.3.10 **Retrospective Search**

2.3.10.1 Purpose

To provide the ability to retrospective search all received message data.

2.3.10.2 Retrospective Search Inputs

Please reference The Jet Propulsion Laboratories API manual for AMHS.

2.3.10.3 Retrospective Search Outputs

Please reference The Jet Propulsion Laboratories API manual for AMHS.

2.3.10.4 CMP Defined API for Retrospective Search Functionality

The CMP defines these API**s** as part of the AMHS. Please reference The Jet Propulsion Laboratories API manual for AMHS. In addition, CMP provides the ability to search the operational Messaging Journal for various criteria using the following API:

```
long rpc_filter_journal
(
  [in,string] char *criList,
  [out] FILTER_LIST **Filter_List
);
```

<u>Parameters</u>

Input

criList	- Filtering criteria

Output

Filter_List	- A list of header information returned based on the input criteria.

Return Values

1	- The returned Filter_List is valid
0	- The call failed to produce a valid Filter_List.

2.3.11 **Operational Journal**

2.3.11.1 Purpose

The operational journal contains a listing of all messages entering of leaving the message processing block. Messages are accessible by a user and location of the message is facilitated through use of a directory. The user is able to sort data in the directory, using multiple sort criteria, retrieve, review, modify, and use this journal as a temporary storage file while composing and/or coordinating a message prior to release. the Operational Journal API consist of:

2.3.11.2 Operational Journal Inputs

See the Journaling Users Manual for a complete list.

2.3.11.3 Operational Journal Outputs

See the Journaling Users Manual for a complete list.

2.3.11.4 CMP Defined API for Operational Journal Functionality

The complete set of Operational Journal API**s** are listed below:

```
long rpc_mark_unmark_delete_entry
(
  [in] long token,
  [in] long revision,
  [in] char *dest
)
```

Parameters

Input -

token	- The token ID of the message.
revision	- The revision number of the message.
dest	- Message destination.
value	- New value for the *deleted* field.

Output -

None

Return Values

1	- Mark/unmark delete entry ok
0	- Mark/unmark delete entry failed

Warning:

At this time, the return value of 1 only means that there was no SQL syntax error. For example, if you try to mark/unmark an entry and the destination you typed in does not exist in the table, it still returns 1 as long there was no SQL syntax error or database error.

<u>Description</u>

The *rpc_mark_unmark_delete_entry()* routine mark/unmarks a message, so the message will (will not) be shown to the specified destination. If the value of *value* is ›Y=, it means the entry will not be shown to that recipient.

long rpc_put_msg

```
(
  [in] HEADER_INFO *header_info,
  [in] char IO,
  [in] TEXT_FILE *Msg
);
```

<u>Parameters</u>

Input -

header_info	- message header information.
IO	- Inbound/outbound message.
Msg	- Message strings.

Output -

None

<u>Return Values</u>

0	- put message failure.
others	- put message ok, the token number is returned.

<u>Description</u>

This function puts a message into the database. The message is stored either in the *Inbound* or *Outbound* directory depend on the value of *IO*. If the value of *IO* is *I*, it means that the message is an inbound message and the message file will be stored in the *Inbound* directory; If the value of *IO* is *O*, it means that the message is an outbound message and the message file will be stored in the *Outbound* directory.

long rpc_put_data

```
(
  [in] long token,
  [in] long revision,
  [in] char IO,
  [in] TEXT_FILE *Data
);
```

<u>Parameters</u>

Input -

token	- Message token ID
revision	- Message revision number
IO	- Inbound/Outbound
Data	- Data

Output -

None

<u>Returned Values</u>

1	- Put data ok.
0	- Put data failure.

<u>Description</u>

The *rpc_put_data()* puts data into the database. The data will be stored either in *Inbound* or *Outbound* Directory depend on the value of *IO*.

```
long rpc_put_status
(
  [in] token,
  [in] revision,
  [in,string] *Status
);
```

<u>Parameters</u>

Input

token	- Message token ID
revision	- Message revision number
Status	- status field value

Output

None

<u>Return Values</u>

1	- put status ok
0	- put status failure

<u>Description</u>

The *rpc_put_status()* routine inserts a status string into the field *status* in the specified message header.

```
long rpc_update_status
(
  [in] long token,
  [in] long revision,
  [in,string]  *Status
);
```

Parameters

Input

token	- Message token ID
revision	- Message revision number
Status	- New message status

Output

None

Return Values

1	- Update status ok
0	- Update status failed

Description

The *rpc_update_status()* routine updates the *status* field in the message header. The value of *status* is updated to *Status.*

```
STATUS rpc_get_status
(
  [in] long token,
  [in] long revision
);
```

Parameters

Input

token	- Message token ID
revision	- Message revision number

Output

None

Return Values

Returns the *status* and *access* field in the message header. Those two fields are combined into a string and is returned.

Description

The *rpc_get_status()* routine returns the status of a message. The status returned combines the *status* field and *access* field.

STATUS rpc_get_user_tag
```
(
  [in] long token,
  [in] long revision
);
```

Parameters

Input

token	- Message token ID
revision	- Message revision number

Output

None

Return Values

0	- get user tag error or the *tag* field is empty.
string	- the *tag* field in the message header.

Description

The *rpc_get_user_tag()* routine returns the *tag* field in the message header in string format.

long rpc_update_msg
```
(
  [in] long token,
  [in]HEADER_INFO *Header_Info,
  [in] char IO,
  [in] TEXT_FILE *Msg
);
```

Parameters

Input

token	- Message token ID
Header_Info	- Message Header
IO	- Inbound/Outbound

Output

None

Return Values

0	- Update message error.
revision #	- The newest revision number of the message.

Description

The *rpc_update_msg()* routine updates a message. Some fields in the message header are also modified.

```
long rpc_display_msg
(
  [in] long token,
  [in] long revision,
  [out]HEADER_INFO **HeaderInfo,
  [out] TEXT_FILE **Msg
);
```

Parameters

Input

token	- Message token ID
revision	- Message revision number

Output

HeaderInfo	- Header info to be send back to the caller.
Msg	- Message to be sent back to the caller.

Return Values

1	- get message ok.
0	- get message failed.

Description

The *rpc_display_msg()* routine returns the message header and the message text back to the caller.

```
long rpc_display_data
(
  [in] long token,
  [in] long revision,
  [out] TEXT_FILE **Ext_Data
);
```

Parameters

Input

token	- Message token ID
revision	- Message revision number

Output

Ext_Data	- data to be sent back to the caller.

Return Values

1	- get data ok
0	- get data failed

Description

The *rpc_display_data()* routine returns the message text data back to the caller.

long rpc_display_audit

```
(
  [in] long token,
  [in] long revision,
  [out] AuditLogType **AuditLog
);
```

Parameters

Input

token	- Message token ID
revision	- Message revision number

Output

AuditLog	- Message audit data to be sent back to the caller.

Return Values

1	- Get message audit data ok.
0	- Get message audit data failed.

Description

The *rpc_display_audit()* routine returns the audit information of a message to the caller.

long rpc_set_user_tag
```
(
  [in] long token,
  [in] long revision,
  [in,string]char *userTag
);
```

__**Parameters**__

Input

token	- Message token ID.
revision	- Message revision number.
userTag	- Tag string.

Output

None

__**Return Values**__

1	- Set user tag ok.
0	- Error in setting user tag.

__**Description**__

The *rpc_set_user_tag()* routine is used to set the *tag* field in a header file.

long rpc_add_memo
```
(
  [in] long token,
  [in] long revision,
  [out] TEXT_FILE *Memo
);
```

__**Parameters**__

Input

token	- Message token ID
revision	- Message revision number

Output

None

__**Return Values**__

1	- Add memo ok.

0	- Error in adding memo.

<u>Description</u>

The *rpc_add_memo()* routine adds a memo to a message header. The *memo* field in the header is set to *Y* to indicate there is a memo file attached to the header.

```
long rpc_delete_memo
(
  [in] long token,
  [in] long revision
);
```

<u>Parameters</u>

Input

token	- Message token ID
revision	- Message revision number

Output

None

<u>Return Values</u>

1	- Delete memo ok
0	- Error in deleting memo

<u>Description</u>

The *rpc_delete_memo()* routine deletes a memo from a message header. The *memo* field in the header is set to *N* to indicate that the memo is deleted.

```
long rpc_modify_memo
(
  [in] long token,
  [in] long revision,
  [in]TEXT_FILE *modifyMemo
);
```

<u>Parameters</u>

Input

token	- Message token ID
revision	- Message revision number

Output

None

Return Values

1	- Modify memo ok
0	- Error in modifying memo

Description

The *rpc_modify_memo()* routine modifies a message memo.

```
long rpc_display_memo
(
  [in] long token,
  [in] long revision,
  [out] TEXT_FILE **Memo
);
```

Parameters

Input

token	- Message token ID
revision	- Message revision number

Output

Memo	- Message memo to be returned.

Return Values

1	- Get message memo ok.
0	- Error in getting message memo.

Description

The *rpc_display_memo()* routine gets a message memo and sends it back to the caller.

```
long rpc_filter_journal
(
  [in,string] char *criList,
  [out] FILTER_LIST **Filter_List
);
```

Parameters

Input

criList	- Filtering criteria

Output

Filter_List	- A list of header information returned based on the input criteria.

Return Values

1	- The criteria is processed ok.
0	- Error in processing filtering criteria.

Description

The *rpc_filter_journal()* routine returns a list of headers based on the input criteria.

Note:

Each returned entry contains only one destination, although a message can have multiple destinations. So if a message to be returned has multiple destinations, multiple message entries will be returned, with each entry contains only one destination. (This way we can mark and not display a message to each specific recipient